Module 13 : Python Fundamentals

Introduction to Python Theory:

1) Introduction to Python and its Features (simple, high-level, interpreted language).

Python is a high-level, interpreted programming language developed by Guido van Rossum in 1991. It is simple, easy to learn, and widely used in areas like web development, data science, and automation. Python's main features include:

- Simple and Readable Syntax – close to English, easy for beginners.
- High-Level Language – no need to manage hardware details.
- Interpreted – executes code line by line for easier debugging.
- Portable and Open Source – runs on multiple platforms and is free to use.
- Object-Oriented and Extensible – supports classes, objects, and large libraries.

Thus, Python is versatile, powerful, and one of the most popular languages today.

2) History and evolution of Python.

Python was developed by Guido van Rossum at Centrum Wiskunde & Informatica (CWI), Netherlands, in the late 1980s and released in **1991** as Python 1.0. It was designed to be simple, readable, and powerful.

- Python 1.0 (1991): Introduced core features like functions, exception handling, and basic data types.
- Python 2.0 (2000): Added list comprehensions, garbage collection, and Unicode support. However, Python 2 eventually became outdated.
- Python 3.0 (2008): Released with major improvements such as better Unicode handling, print as a function, and removal of old constructs, but it was not backward compatible with Python 2.

- Today: Python 3.x is the standard, with continuous updates adding features like type hints, async programming, and improved performance.

Python has grown into one of the most popular languages in the world, widely used in web development, data science, artificial intelligence, automation, and more.

3) Advantages of using Python over other programming languages.

1. Easy to Learn and Use – Python has simple, English-like syntax, making it beginner-friendly.
2. Cross-Platform – Python code runs on different operating systems without changes.
3. Large Standard Library – Provides built-in modules for tasks like math, file handling, and web development.
4. Faster Development – Short and readable code reduces development time compared to languages like C++ or Java.
5. Versatile – Supports web development, data science, AI, automation, and more.
6. Strong Community Support – Huge community with libraries, frameworks, and resources.

Conclusion: Python's simplicity, flexibility, and wide applications make it more advantageous than many other programming languages.

4) Installing Python and setting up the development environment (Anaconda, PyCharm, or VS Code).

Installing Python and Setting up the Development Environment

To start programming in Python, we first need to install it and set up a suitable development environment.

1. Installing Python:

   Download Python from the official website python.org.

   Run the installer, check "Add Python to PATH," and complete installation.

Verify installation by typing python --version in the command prompt/terminal.

2. Using Anaconda**:**

Anaconda is a distribution of Python that comes with many pre-installed libraries for data science and machine learning.

It includes Jupyter Notebook, which is very useful for interactive coding.

3. Using PyCharm:

PyCharm is a popular IDE (Integrated Development Environment) for Python.

It provides features like code completion, debugging, and project management.

4. Using VS Code**:**

Visual Studio Code is a lightweight editor that supports Python through extensions.

It is flexible, fast, and widely used by developers.

Conclusion**:** Python can be installed easily, and developers can choose Anaconda, PyCharm, or VS Code based on their needs for learning or professional work.

5) Writing and executing your first Python program

After installing Python, you can write and run your first program.

1. Writing the Program**:**

Open any text editor or IDE (like IDLE, VS Code, or PyCharm).

Type the following code:

print("Hello, World!")

2. Saving the File**:**

Save the file with the extension .py, for example hello.py.

3. Executing the Program:

Open the command prompt/terminal.

Navigate to the folder where the file is saved.

Run the program using:

python hello.py

4. Output:

Hello, World!

2. Programming Style

6) Understanding Python's PEP 8 guidelines

PEP 8 (Python Enhancement Proposal 8) is the official style guide for writing clean and readable Python code. It provides rules and recommendations on how to format Python programs so that the code looks consistent and easy to understand.

Key points of PEP 8 include:

- Indentation: Use 4 spaces per indentation level.
- Line Length: Limit lines to 79 characters.
- Naming Conventions: Use snake_case for variables and functions, CamelCase for classes, and UPPER_CASE for constants.
- Spacing: Add spaces around operators and after commas, but not inside brackets.
- Comments: Write clear and meaningful comments to explain code.
- Imports: Place all import statements at the top of the file.

7) Indentation, comments, and naming conventions in Python.

1 Indentation

- Indentation means adding spaces at the beginning of a line to define code blocks.
- In Python, indentation is **mandatory** (usually 4 spaces).

- Example:

  if True:

    print("Indented block")

2 Comments

- Comments are notes in the code that Python ignores during execution.
- Used to explain code for better understanding.

  Single-line comment:

    # This is a single-line comment

  Multi-line comment (using triple quotes):

    """This is

    a multi-line comment"""

3 Naming Conventions

- Follow PEP 8 guidelines for naming:
  - Variables and functions: snake_case (e.g., student_name)
  - Classes: CamelCase (e.g., StudentData)
  - Constants: UPPER_CASE (e.g., PI = 3.14)

8) Writing readable and maintainable code.

Readable and maintainable code is code that is easy to understand, modify, and reuse. In Python, this can be achieved by:

- Using proper indentation and formatting to make code structure clear.
- Following PEP 8 guidelines for naming conventions, line length, and spacing.
- Writing meaningful names for variables, functions, and classes.
- Adding comments and docstrings to explain the purpose of code.
- Breaking code into functions and modules for reusability and organization.

- Avoiding complexity by keeping code simple and clear.

3. Core Python Concepts

9) Understanding data types: integers, floats, strings, lists, tuples, dictionaries, sets.

1 Integers (int)

- Represent whole numbers (positive or negative) without decimals.
- Example: x = 25, y = -10
- Commonly used in counting, indexing, and arithmetic operations.

2 Floats (float)

- Represent real numbers with decimal points.
- Example: pi = 3.14, marks = 89.5
- Useful for scientific calculations and precise measurements.

3 Strings (str)

- Sequence of characters enclosed in single ('), double (") or triple quotes.
- Example: name = "Python", msg = 'Hello'
- Strings support operations like slicing, concatenation, and formatting.

4 Lists (list)

- Ordered, mutable (can be changed) collections of items.
- Example: fruits = ["apple", "banana", "mango"]
- Allow indexing, iteration, and methods like append(), remove(), etc.

5 Tuples (tuple)

- Ordered, immutable (cannot be changed) collections of items.
- Example: coordinates = (10, 20, 30)
- Faster than lists and used for fixed data.

6 Dictionaries (dict)

- Store data as key–value pairs.
- Example: student = {"name": "John", "age": 21}
- Keys must be unique, and values can be of any type.
- Used for mapping and fast lookups.

7 Sets (set)

- Unordered collections of unique items.
- Example: numbers = {1, 2, 3, 4, 4} → {1, 2, 3, 4} (duplicates removed)
- Useful for mathematical operations like union, intersection, and difference.

10) Python variables and memory allocation.

Variables in Python

- A variable is a name given to a memory location that stores data.
- In Python, variables are created automatically when you assign a value.
- Example:

```
x = 10      # integer variable
name = "Raj"  # string variable
```

- Python is dynamically typed, meaning you don't need to declare the type of a variable explicitly; it is decided at runtime.

Memory Allocation in Python

- When a variable is created, Python allocates memory to store the object (value).
- The variable name acts as a reference (or pointer) to that memory location.
- Python uses an internal mechanism called Object Reference Model:
    - Each object has a unique id (memory address), type, and value.
    - Example:

x = 10

y = x

- print(id(x), id(y)) # Both point to the same memory for value 10

- Python uses automatic memory management, which includes:
    - Reference Counting**:** Keeps track of how many variables refer to an object.
    - Garbage Collection**:** Automatically frees memory when an object is no longer used.

3 Important Points

- Same values may point to the same memory location (object reusability).
- Mutable objects (like lists, dicts) can be changed in place, while immutable objects (like int, float, string, tuple) create new memory when changed.

11) Python operators: arithmetic, comparison, logical, bitwise

Operators in Python are special symbols that perform operations on variables and values. They are used to carry out calculations and logical decisions.

1. Arithmetic Operators

Used for basic mathematical operations.

Operator	Description	Example	Output
+	Addition	10 + 5	15
-	Subtraction	10 - 5	5
*	Multiplication	10 * 5	50
/	Division	10 / 5	2.0
//	Floor Division	10 // 3	3
%	Modulus (Remainder)	10 % 3	1
**	Exponent (Power)	2 ** 3	8

2. Comparison (Relational) Operators

Used to compare values, return result as True or False

Operator	Description	Example	Output
==	Equal to	5 == 5	True
!=	Not equal to	5 != 3	True
>	Greater than	10 > 5	True
<	Less than	2 < 5	True
>=	Greater or equal	5 >= 5	True
<=	Less or equal	3 <= 5	True

3. Logical Operators

Used to combine conditional statements.

Operator	Description	Example	Output
and	True if both conditions are true	(5 > 2 and 10 > 5)	True
or	True if at least one condition is true	(5 > 10 or 3 < 8)	True
not	Reverses the result	not(5 > 2)	False

4. Bitwise Operators

Used for operations on binary (bit-level) data

Operator	Description	Example	Output
&	Bitwise AND	5 & 3	1

Operator	Description	Example	Output
`	`	Bitwise OR	5 | 3
^	Bitwise XOR	5 ^ 3	6
~	Bitwise NOT	~5	-6
<<	Left shift	5 << 1	10
>>	Right shift	5 >> 1	2

4. Conditional Statements

12) Introduction to conditional statements: if, else, elif.

Conditional statements in Python are used to make decisions in a program based on certain conditions. They control the flow of execution by running different blocks of code depending on whether a condition is True or False.

1 if Statement

- Executes a block of code only if the condition is true.

```
age = 18
if age >= 18:
    print("You are eligible to vote")
```

2 else Statement

- Executes a block of code if the if condition is false.

```
age = 16
if age >= 18:
```

```
    print("You are eligible to vote")
else:
    print("You are not eligible to vote")
```

3 elif Statement (else if)

- Used to check multiple conditions one by one.

```
marks = 75
if marks >= 90:
    print("Grade A")
elif marks >= 60:
    print("Grade B")
else:
    print("Grade C")
```

13) Nested if-else conditions.

In Python, nested if-else means using one if-else statement inside another. It allows checking multiple conditions in a structured way, where a decision depends on another decision.

Syntax

```
if condition1:
    if condition2:
        # code block if both conditions are true
    else:
        # code block if condition1 is true but condition2 is false
else:
    # code block if condition1 is false
```

Example

```
age = 20
```

```
if age >= 18:
    if age >= 21:
        print("You can vote and also drink (if allowed)")
    else:
        print("You can vote but not drink")
else:
    print("You are not eligible to vote")
```

5. Looping (For, While)

14)Introduction to for and while loops.

Loops in Python are used to repeat a block of code multiple times until a condition is met. They help avoid writing repetitive code. Python provides two main **types of loops:**

1. **For Loop**

Used to iterate over a sequence like a list, tuple, string, or range of numbers.

The number of iterations is known or fixed.

Syntax:

```
for variable in sequence:
    #code block
```

Example:

```
for i in range(5):
    print(i)
```

Output:

0

1

2

3

4

2) While Loop

- Repeats a block of code as long as a condition is True.
- The number of iterations may not be fixed.

Syntax:

```
while condition:
    # code block
```

Example:

```
count = 0
while count < 5:
    print(count)
    count += 1
```

Output:

0

1

2

3

4

15) How loops work in Python.

Loops in Python are used to repeat a block of code multiple times until a certain condition is met. Python primarily uses for loops and while loops.

1. For Loop:

   Iterates over a sequence (like list, tuple, string, or range).

   At each iteration, the loop variable takes the next value from the sequence.

The loop automatically stops when all elements in the sequence are processed.

Example:

```
for i in range(3):
    print(i)
```

Output:

0

1

2

2. While Loop:

Repeats the code as long as a condition is True.

Before each iteration, Python checks the condition. If it is False, the loop stops.

Example:

```
count = 0
while count < 3:
    print(count)
    count += 1
```

Output:

0

1

2

3 Loop Flow:

Initialization**:** Set starting values or variables.

Condition Check**:** Python evaluates the loop condition.

Execution**:** If condition is True, executes the loop body.

Update**:** Updates loop variables (for while loops, manually; for for loops, automatically).

Termination: Loop stops when condition becomes False or sequence ends

6. Generators and Iterators

16 Understanding how generators work in Python.

Generators in Python are a type of iterable that allow you to iterate over data one item at a time without storing the entire sequence in memory. They are useful for handling large datasets efficiently.

1. How Generators Work:

   Generators are created using functions with the yield statement instead of return.

   Each time yield is called, it produces a value and pauses the function, saving its state for the next call.

   When the generator is called again, it resumes from where it left off.

Example

```
def simple_generator():
  for i in range(3):
    yield i


gen = simple_generator()
print(next(gen))  # Output: 0
print(next(gen))  # Output: 1
print(next(gen))  # Output: 2
```

3 Advantages of Generators:

- Memory-efficient**:** Only one value is produced at a time.
- Lazy evaluation: Values are generated on-the-fly.

- Useful for large datasets or streams where storing all items is not practical.

17) Difference between yield and return.

Feature	return	yield
Purpose	Returns a value from a function and terminates the function.	Returns a value from a generator and pauses the function, saving its state.
Function Type	Used in normal functions.	Used in generator functions.
Memory Usage	Returns all values at once (memory used depends on all returned data).	Produces values one at a time (memory-efficient).
Execution	Function ends after return.	Function can resume after yield when next value is requested.
Iteration	Cannot be iterated over; single return per call.	Can be iterated over multiple times using next() or a loop.

Example	def f():
	  return 5
		def g():
		  yield 5

18) Understanding iterators and creating custom iterators.

1. What is an Iterator?

- An iterator is an object in Python that allows us to traverse through all the elements of a collection (like a list, tuple, or set) one at a time.

Iterators implement two methods:

- iter() → Returns the iterator object itself.
- next() → Returns the next element in the sequence. Raises StopIteration when the sequence ends.

Example with built-in iterator:

```
my_list = [1, 2, 3]
it = iter(my_list)
print(next(it))  # Output: 1
print(next(it))  # Output: 2
print(next(it))  # Output: 3
```

Creating Custom Iterators Without OOP

Explanation:

- A generator function is a function that uses the yield keyword to produce values one at a time.
- Every time yield is called, the function pauses and remembers its state.
- When the generator is called again (with next() or in a loop), it resumes from where it left off.
- This makes generator functions a simple way to create custom iterators without classes.

Example: Even Numbers Iterator

```
def even_numbers(max_value):
    num = 2
    while num <= max_value:
        yield num
        num += 2

# Using the generator
evens = even_numbers(10)
for n in evens:
```

```
    print(n)
```

Output:

2

4

6

8

10

19) Defining and calling functions in Python.

1. What is a Function?

- A function is a block of reusable code that performs a specific task.
- Functions help avoid repetition, make code organized, and improve readability.

2. Defining a Function

- Use the def keyword to define a function

Syntax:

```
def function_name(parameters):
    # code block
    return value  # optional
```

Example

```
def greet(name):
    print("Hello, " + name + "!")
```

3. Calling a Function

- To execute the function, use its name followed by parentheses and pass required arguments.

```
greet("Alice")
```

Output:

Hello, Alice!

4. Key Points:

- Parameters/Arguments**:** Values passed to the function for processing.
- Return Statement**:** Sends a result back to the caller. If not used, the function returns None by default.
- Functions can be called multiple times with different arguments.

20) Function arguments (positional, keyword, default).

Python functions can accept different types of arguments to make them flexible. The main types are positional, keyword, and default arguments.

1. Positional Arguments

- Arguments are passed to the function in the same order as the parameters.
- Example:

```
def greet(name, age):
    print(f"Hello {name}, you are {age} years old.")


greet("Alice", 25)  # Output: Hello Alice, you are 25 years old.
```

2. Keyword Arguments

- Arguments are passed using parameter names, so order does not matter.
- Example:

```
greet(age=25, name="Alice")  # Output: Hello Alice, you are 25 years old.
```

3. Default Arguments

- Parameters can have default values. If no argument is provided, the default is used.
- Example:

```
def greet(name, age=18):
    print(f"Hello {name}, you are {age} years old.")


greet("Bob")       # Output: Hello Bob, you are 18 years old.
greet("Alice", 25)  # Output: Hello Alice, you are 25 years old.
```

21) Scope of variables in Python.

The scope of a variable determines where in the program the variable can be accessed or modified. Python has different types of variable scopes:

1. Local Scope

- A variable declared inside a function is local to that function.
- It can only be accessed within the function.

Example:

```
def my_func():
    x = 10  # Local variable
    print(x)


my_func()    # Output: 10
# print(x)   # Error: x is not defined outside the function
```

2. Global Scope

- A variable declared outside all functions is global.
- It can be accessed anywhere in the program.

Example:

```
y = 20  # Global variable


def my_func():
```

```
    print(y)


my_func()  # Output: 20
print(y)   # Output: 20
```

3. Enclosing (Nonlocal) Scope

- Occurs in nested functions.
- Inner function can access variables from the enclosing outer function using nonlocal.
- Example:

```
def outer():
    z = 5
    def inner():
        nonlocal z
        z += 1
        print(z)
    inner()


outer()  # Output: 6
```

4. Built-in Scope

- Refers to names pre-defined in Python, like print(), len(), etc.
- These can be used anywhere in the program without declaration.

22) Built-in methods for strings, lists, etc.

Method	Description	Example	Output
upper()	Converts all characters to uppercase	"hello".upper()	"HELLO"

Method	Description	Example	Output
lower()	Converts all characters to lowercase	"HELLO".lower()	"hello"
strip()	Removes leading/trailing spaces	" hello ".strip()	"hello"
replace()	Replaces a substring with another	"hello".replace("h","H")	"Hello"
split()	Splits string into a list of words	"a b c".split()	['a', 'b', 'c']

2. List Methods

Method	Description	Example	Output
append()	Adds an element at the end	[1,2].append(3)	[1,2,3]
extend()	Adds multiple elements	[1,2].extend([3,4])	[1,2,3,4]
insert()	Inserts an element at a specific position	[1,2].insert(1,5)	[1,5,2]
remove()	Removes a specific element	[1,2,3].remove(2)	[1,3]
pop()	Removes and returns last element	[1,2,3].pop()	[1,2] (returns 3)
sort()	Sorts the list	[3,1,2].sort()	[1,2,3]

3. Dictionary Methods

Method	Description	Example	Output
keys()	Returns all keys	{"a":1,"b":2}.keys()	dict_keys(['a','b'])
values()	Returns all values	{"a":1,"b":2}.values()	dict_values([1,2])
items()	Returns key-value pairs	{"a":1}.items()	dict_items([('a',1)])
get()	Returns value of a key	{"a":1}.get("a")	1

Method	Description	Example	Output
update()	Adds or updates key-value pairs	{"a":1}.update({"b":2})	{'a':1,'b':2}

8. Control Statements (Break, Continue, Pass)

23) Understanding the role of break, continue, and pass in Python loops

1. break

Exits the loop completely when the condition is met.

```
for i in range(1, 6):
    if i == 3:
        break   # Exit loop when i = 3
    print(i)
```

Output:

1

2

As soon as i == 3, the loop stops.

2. continue

Skips the current iteration and jumps to the next iteration of the loop.

```
for i in range(1, 6):
    if i == 3:
        continue   # Skip this iteration
    print(i)
```

Output:

1

2

4

5

i == 3 is skipped, but the loop continues for the rest.

3. pass

Does nothing. It’s just a placeholder (useful when you don’t want to write code yet).

```
for i in range(1, 6):
    if i == 3:
        pass   # Does nothing
    print(i)
```

Output:

1

2

3

4

5

Even though i == 3 triggers pass, the loop still runs normally.

9. String Manipulation

24) Understanding how to access and manipulate strings.

1. Accessing Strings

A string is a sequence of characters. We can access them using **indexing** and slicing.

Indexing

- text[i] → returns character at position i.

- Positive index starts from 0 (left → right).
- Negative index starts from -1 (right → left).

```
text = "Python"

print(text[0])   # P   → first character
print(text[3])   # h   → 4th character
print(text[-1])  # n   → last character
```

Slicing

- text[start:end:step] → extracts substring.
- Includes start, excludes end.
- step defines the jump size.

```
text = "Python"

print(text[0:4])   # Pyth   → characters 0 to 3
print(text[:3])    # Pyt    → from start to 2
print(text[2:])    # thon   → from index 2 to end
print(text[::-1])  # nohtyP → reversed string
```

2. Manipulating Strings

Python has many built-in string methods.

Changing case

- upper() → Converts all characters to uppercase.
- lower() → Converts all characters to lowercase.
- title() → Capitalizes the first letter of each word.

- capitalize() → Capitalizes the first letter of the string.

```
s = "hello world"
print(s.upper())       # HELLO WORLD
print(s.lower())       # hello world
print(s.title())       # Hello World
print(s.capitalize())  # Hello world
```

Removing spaces

- strip() → Removes spaces (both sides).
- lstrip() → Removes spaces from left side.
- rstrip() → Removes spaces from right side.

```
s = "   Python   "
print(s.strip())   # "Python"
print(s.lstrip())  # "Python   "
print(s.rstrip())  # "   Python"
```

Searching and Replacing

- find(sub) → Returns index of first occurrence of substring (or -1 if not found).
- replace(old, new) → Replaces all occurrences of old with new.

```
s = "I love Python"
print(s.find("Python"))         # 7
print(s.replace("Python","Java"))  # I love Java
```

Splitting and Joining

- split(sep) → Splits string into list using separator.
- join(list) → Joins elements of list into a string with separator.

```
s = "apple,banana,cherry"
fruits = s.split(",")   # ['apple', 'banana', 'cherry']
print(fruits)

joined = "-".join(fruits)   # apple-banana-cherry
print(joined)
```

Checking Content

- isalpha() → True if only letters.
- isdigit() → True if only digits.
- isalnum() → True if only letters + digits.

```
s = "Python123"
print(s.isalpha())     # False (letters + numbers)
print(s.isdigit())     # False
print("123".isdigit()) # True
print(s.isalnum())     # True
```

String Formatting

- f"{}" → f-strings allow inserting variables inside text.

```
name = "Alice"
age = 25

print(f"My name is {name}, I am {age} years old.")
```

My name is Alice, I am 25 years old.

25) Basic operations: concatenation, repetition, string methods (upper(), lower(), etc.

Basic String Operations

1. Concatenation (+)

Joins two or more strings together.

s1 = "Hello"

s2 = "World"

print(s1 + " " + s2)

Output:

Hello World

2. Repetition (*)

Repeats the string multiple times.

word = "Hi "

print(word * 3)

Output:

Hi Hi Hi

3. String Methods

Case Conversion

- upper() → Converts to uppercase.
- lower() → Converts to lowercase.
- title() → First letter of each word capitalized.
- capitalize() → First letter of string capitalized.

- swapcase() → Swap uppercase ↔ lowercase.

```
s = "hello world"
print(s.upper())       # HELLO WORLD
print(s.lower())       # hello world
print(s.title())       # Hello World
print(s.capitalize())  # Hello world
print(s.swapcase())    # HELLO WORLD -> hello world / vice versa
```

Strip Spaces

- strip() → Remove spaces (both sides).
- lstrip() → Remove left spaces.
- rstrip() → Remove right spaces.

```
s = "   Python   "
print(s.strip())   # Python
```

Search & Replace

- find(sub) → Returns index of first occurrence.
- replace(old, new) → Replace substring.

```
s = "I love Python"
print(s.find("Python"))       # 7
print(s.replace("Python", "Java"))  # I love Java
```

Split & Join

- split(sep) → Break string into list.
- join(list) → Join list elements with separator.

```
s = "apple,banana,cherry"
fruits = s.split(",")     # ['apple', 'banana', 'cherry']
print("-".join(fruits))   # apple-banana-cherry
```

Check Content

- isalpha() → True if only letters.
- isdigit() → True if only numbers.
- isalnum() → True if letters + numbers.

```
s = "Python123"
print(s.isalpha())  # False
print(s.isdigit())  # False
print(s.isalnum())  # True
```

26) String slicing.

String Slicing

In Python, slicing means extracting parts of a string using:

string[start:end:step]

- start → index where slice begins (inclusive).
- end → index where slice stops (exclusive).
- step → how many characters to skip. (default = 1)

Examples

```
text = "Python"
```

1. Basic Slicing

```
print(text[0:4])   # Pyth  → from index 0 to 3
print(text[:3])    # Pyt   → from start to 2
```

print(text[2:]) # thon → from index 2 to end

2. Negative Indexing

Negative indices count from the end.

print(text[-6:-1]) # Pytho → from -6 (P) to -2 (o)

print(text[-3:]) # hon → last 3 characters

print(text[:-3]) # Pyt → from start to -4

3. Using Step

The third argument skips characters.

print(text[::2]) # Pto → every 2nd character

print(text[1::2]) # yhn → every 2nd char starting at index 1

4. Reversing a String

Step = -1 reverses the string.

print(text[::-1]) # nohtyP

10. Advanced Python (map(), reduce(), filter(), Closures and Decorators)

27) How functional programming works in Python.

Functional Programming in Python

Functional Programming is a style where you focus on **functions** (pure, reusable, independent of state) instead of changing variables or using loops heavily.

Python supports FP concepts through functions, higher-order functions, immutability, and built-in FP tools.

1. Pure Functions

A function that always gives the same output for the same input, without side effects.

```
def add(a, b):
    return a + b   # No external dependency, no variable modification
```

Predictable, easy to test.

2. First-Class Functions

In Python, functions are treated like objects. You can:

- Assign to a variable
- Pass as an argument
- Return from another function

```
def square(x):
    return x * x


func = square   # assign function
print(func(5))  # 25
```

3. Higher-Order Functions

A function that takes other functions as arguments or returns one.

Example: map(), filter(), reduce()

```
nums = [1, 2, 3, 4, 5]


# map → apply function to each element
squares = list(map(lambda x: x**2, nums))
print(squares)   # [1, 4, 9, 16, 25]
```

```
# filter → keep elements that satisfy condition
evens = list(filter(lambda x: x % 2 == 0, nums))
print(evens)     # [2, 4]

# reduce → apply rolling calculation
from functools import reduce
sum_all = reduce(lambda a, b: a + b, nums)
print(sum_all)   # 15
```

4. Immutability

In Function Programming, data should not be changed (avoid side effects). Instead of modifying a list, create a new one.

```
nums = [1, 2, 3]
new_nums = [x * 2 for x in nums]  # does not change original
print(nums)      # [1, 2, 3]
print(new_nums)  # [2, 4, 6]
```

5. Anonymous Functions (Lambda)

Small one-line functions without def.

```
double = lambda x: x * 2
print(double(10))  # 20
```

6. Recursion

Instead of loops, FP often uses recursion.

```
def factorial(n):
    if n == 0 :
        return 1
    else:
        return n * factorial(n-1)

print(factorial(5))
```

28) Using map(), reduce(), and filter() functions for processing data.

Python provides map(), filter(), reduce() for functional-style data transformation.

1. map(function, iterable)

Applies a function to each element of an iterable. Returns a map object (convert to list() to see results).

```
nums = [1, 2, 3, 4, 5]

# square each number
squares = list(map(lambda x: x**2, nums))
print(squares)   # [1, 4, 9, 16, 25]
```

Used when you want to transform all elements.

2. filter(function, iterable)

Keeps only elements where the function returns True.

```
nums = [1, 2, 3, 4, 5, 6]

# keep even numbers
```

```
evens = list(filter(lambda x: x % 2 == 0, nums))

print(evens)   # [2, 4, 6]
```

Used when you want to select a subset of elements.

3. reduce(function, iterable)

Repeatedly applies a function to elements, reducing them to a single value. Must import from functools.

```
from functools import reduce


nums = [1, 2, 3, 4, 5]


# sum of all numbers

total = reduce(lambda a, b: a + b, nums)

print(total)   # 15
```

Used when you want to aggregate values (sum, product, max, etc.).

→ Combined Example

```
from functools import reduce


nums = [1, 2, 3, 4, 5, 6]


# Step 1: square all numbers (map)

squares = list(map(lambda x: x**2, nums))     # [1, 4, 9, 16, 25, 36]


# Step 2: filter even squares (filter)
```

```
even_squares = list(filter(lambda x: x % 2 == 0, squares))   # [4, 16, 36]


# Step 3: sum of even squares (reduce)
result = reduce(lambda a, b: a + b, even_squares)   # 56


print(result)
```

Final Output → 56

29) Closures and Decorators in Python

1. Closures

0A closure is a function defined inside another function that remembers variables from the outer function, even after the outer function has finished execution.

Example:

```
def outer(msg):
    def inner():
        print("Message:", msg)  # inner remembers 'msg'
    return inner


greet = outer("Hello Python")
greet()   # Message: Hello Python
```

Explanation:

- outer("Hello Python") returns inner.
- inner() still remembers msg = "Hello Python" → this is a closure.

Use cases: data hiding, callbacks, factory functions.

2. Decorators

A decorator is a special function that modifies the behavior of another function without changing its code.

- They are built on top of closures.
- In Python, decorators are often used for logging, authentication, performance checks, etc.

Example of a Decorator:

```
def decorator(func):
    def wrapper():
        print("Before function call")
        func()
        print("After function call")
    return wrapper


@decorator   # same as: hello = decorator(hello)
def hello():
    print("Hello World!")


hello()
```

Output:

Before function call

Hello World!

After function call

@decorator modifies hello() by wrapping it with extra behavior.